boolean.py Documentation

Release 3.2

Sebastian Krämer

Aug 06, 2018

Contents

1 User Guide **1**
- 1.1 Introduction . . . 1
- 1.2 Installation . . . 2
- 1.3 Creating boolean expressions . . . 2
- 1.4 Evaluation of expressions . . . 2
- 1.5 Equality of expressions . . . 3
- 1.6 Analyzing a boolean expression . . . 4
- 1.7 Using boolean.py to define your own boolean algebra . . . 5

2 Concepts and Definitions **7**
- 2.1 Basic Definitions . . . 7
- 2.2 Laws . . . 9

3 Development Guide **13**
- 3.1 Testing . . . 13
- 3.2 Classes Hierarchy . . . 14
- 3.3 Class creation . . . 14
- 3.4 Class initialization . . . 14
- 3.5 Ordering . . . 14
- 3.6 Parsing . . . 15

4 Acknowledgments **17**

CHAPTER 1

User Guide

This document provides an introduction on **boolean.py** usage. It requires that you are already familiar with Python and know a little bit about boolean algebra. All definitions and laws are stated in *Concepts and Definitions*.

Contents

- *User Guide*
  - *Introduction*
  - *Installation*
  - *Creating boolean expressions*
  - *Evaluation of expressions*
  - *Equality of expressions*
  - *Analyzing a boolean expression*
  - *Using boolean.py to define your own boolean algebra*

1.1 Introduction

boolean.py implements a boolean algebra. It defines two base elements, *TRUE* and *FALSE*, and a class `Symbol` for variables. Expressions are built by composing symbols and elements with AND, OR and NOT. Other compositions like XOR and NAND are not implemented.

1.2 Installation

```
pip install boolean.py
```

1.3 Creating boolean expressions

There are three ways to create a boolean expression. They all start by creating an algebra, then use algebra attributes and methods to build expressions.

You can build an expression from a string:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> algebra.parse('x & y')
AND(Symbol('x'), Symbol('y'))

>>> parse('(apple or banana and (orange or pineapple and (lemon or cherry)))')
OR(Symbol('apple'), AND(Symbol('banana'), OR(Symbol('orange'), AND(Symbol('pineapple
↪'), OR(Symbol('lemon'), Symbol('cherry'))))))
```

You can build an expression from a Python expression:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y = algebra.symbols('x', 'y')
>>> x & y
AND(Symbol('x'), Symbol('y'))
```

You can build an expression by using the algebra functions:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y = algebra.symbols('x', 'y')
>>> TRUE, FALSE, NOT, AND, OR, symbol = algebra.definition()
>>> expr = AND(x, y, NOT(OR(symbol('a'), symbol('b'))))
>>> expr
AND(Symbol('x'), Symbol('y'))
>>> print(expr.pretty())

>>> print(expr)
```

1.4 Evaluation of expressions

By default, an expression is not evaluated. You need to call the `simplify()` method explicitly an expression to perform some minimal simplification to evaluate an expression:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y = algebra.symbols('x', 'y')
>>> print(x&~x)
0
>>> print(x|~x)
```

(continues on next page)

(continued from previous page)

```
1
>>> print(x|x)
x
>>> print(x&x)
x
>>> print(x&(x|y))
x
>>> print((x&y) | (x&~y))
x
```

When simplify() is called, the following boolean logic laws are used recursively on every sub-term of the expression:

- *Associativity*
- *Annihilator*
- *Idempotence*
- *Identity*
- *Complementation*
- *Elimination*
- *Absorption*
- *Negative absorption*
- *Commutativity* (for sorting)

Also double negations are canceled out (*Double negation*).

A simplified expression is return and may not be fully evaluated nor minimal:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y, z = algebra.symbols('x', 'y', 'z')
>>> print((((x|y)&z)|x&y).simplify())
(x&y)|(z&(x|y))
```

1.5 Equality of expressions

The expressions equality is tested by the `__eq__()` method and therefore the output of $expr_1 == expr_2$ is not the same as mathematical equality.

Two expressions are equal if their structure and symbols are equal.

1.5.1 Equality of Symbols

Symbols are equal if they are the same or their associated objects are equal.

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y, z = algebra.symbols('x', 'y', 'z')
>>> x == y
False
>>> x1, x2 = algebra.symbols("x", "x")
```

(continues on next page)

(continued from previous page)

```
>>> x1 == x2
True
>>> x1, x2 = algebra.symbols(10, 10)
>>> x1 == x2
True
```

1.5.2 Equality of structure

Here are some examples of equal and unequal structures:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> expr1 = algebra.parse("x|y")
>>> expr2 = algebra.parse("y|x")
>>> expr1 == expr2
True
>>> expr = algebra.parse("x|~x")
>>> expr == TRUE
False
>>> expr1 = algebra.parse("x&(~x|y)")
>>> expr2 = algebra.parse("x&y")
>>> expr1 == expr2
False
```

1.6 Analyzing a boolean expression

1.6.1 Getting sub-terms

All expressions have a property `args` which is a tuple of its terms. For symbols and base elements this tuple is empty, for boolean functions it contains one or more symbols, elements or sub-expressions.

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> algebra.parse("x|y|z").args
(Symbol('x'), Symbol('y'), Symbol('z'))
```

1.6.2 Getting all symbols

To get a set() of all unique symbols in an expression, use its `symbols` attribute

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> algebra.parse("x|y&(x|z)").symbols
{Symbol('y'), Symbol('x'), Symbol('z')}
```

To get a list of all symbols in an expression, use its `get_symbols` method

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> algebra.parse("x|y&(x|z)").get_symbols()
[Symbol('x'), Symbol('y'), Symbol('x'), Symbol('z')]
```

1.6.3 Literals

Symbols and negations of symbols are called literals. You can test if an expression is a literal:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y, z = algebra.symbols('x', 'y', 'z')
>>> x.isliteral
True
>>> (~x).isliteral
True
>>> (x|y).isliteral
False
```

Or get a set() or list of all literals contained in an expression:

```
>>> import boolean
>>> algebra = boolean.BooleanAlgebra()
>>> x, y, z = algebra.symbols('x', 'y', 'z')
>>> x.literals
{Symbol('x')}
>>> (~(x|~y)).get_literals()
[Symbol('x'), NOT(Symbol('y'))]
```

To remove negations except in literals use `literalize()`:

```
>>> (~(x|~y)).literalize()
~x&y
```

1.6.4 Substitutions

To substitute parts of an expression, use the `subs()` method:

```
>>> e = x|y&z
>>> e.subs({y&z:y})
x|y
```

1.7 Using boolean.py to define your own boolean algebra

You can customize about everything in boolean.py to create your own custom algebra: 1. You can subclass `BooleanAlgebra` and override or extend the `tokenize()` and `parse()` methods to parse custom expressions creating your own mini expression language. See the tests for examples.

2. You can subclass the Symbol, NOT, AND and OR functions to add additional methods or for custom representations. When doing so, you configure `BooleanAlgebra` instances by passing the custom sub-classes as agruments.

CHAPTER 2

Concepts and Definitions

In this document the basic definitions and important laws of Boolean algebra are stated.

Contents

- *Concepts and Definitions*
  - *Basic Definitions*
  - *Laws*

2.1 Basic Definitions

2.1.1 Boolean Algebra

This is the main entry point. An algebra is define by the actual classes used for its domain, functions and variables.

2.1.2 Boolean Domain

S := {1, 0}

These base elements are algebra-level singletons classes (only one instance of each per algebra instance), called `TRUE` *and* `FALSE`.

2.1.3 Boolean Variable

A variable holds an object and its implicit value is TRUE.

Implemented as class or subclasses of class `Symbol`.

2.1.4 Boolean Function

A function $f : S^n \rightarrow S$ (where n is called the order).

Implemented as class `Function`.

2.1.5 Boolean Expression

Either a base element, a boolean variable or a boolean function.

Implemented as class `Expression` *- this is the base class for* `BaseElement`, `Symbol` *and* `Function`.

2.1.6 NOT

A boolean function of order 1 with following truth table:

x	NOT(x)
0	1
1	0

Instead of $NOT(x)$ one can write $\sim x$.

Implemented as class `NOT`.

2.1.7 AND

A boolean function of order 2 or more with the truth table for two elements

x	y	AND(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

and the property $AND(x, y, z) = AND(x, AND(y, z))$ where x, y, z are boolean variables.

Instead of $AND(x, y, z)$ one can write $x\&y\&z$.

Implemented as class `AND`.

2.1.8 OR

A boolean function of order 2 or more with the truth table for two elements

x	y	OR(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

and the property $OR(x, y, z) = OR(x, OR(y, z))$ where x, y, z are boolean expressions.

Instead of $OR(x, y, z)$ one can write $x|y|z$.

Implemented as class `OR`.

2.1.9 Literal

A boolean variable, base element or its negation with NOT.

Implemented indirectly as `Expression.isliteral`, `Expression.literals` *and* `Expression.literalize()`.

2.1.10 Disjunctive normal form (DNF)

A disjunction of conjunctions of literals where the conjunctions don't contain a boolean variable *and* it's negation. An example would be $x\&y|x\&z$.

Implemented as `BooleanAlgebra.dnf`.

2.1.11 Full disjunctive normal form (FDNF)

A DNF where all conjunctions have the same count of literals as the whole DNF has boolean variables. An example would be $x\&y\&z|x\&y\&(\sim z)|x\&(\sim y)\&z$.

2.1.12 Conjunctive normal form (CNF)

A conjunction of disjunctions of literals where the disjunctions don't contain a boolean variable *and* it's negation. An example would be $(x|y)\&(x|z)$.

Implemented as `BooleanAlgebra.cnf`.

2.1.13 Full conjunctive normal form (FCNF)

A CNF where all disjunctions have the same count of literals as the whole CNF has boolean variables. An example would be: $(x|y|z)\&(x|y|(\sim z))\&(x|(\sim y)|z)$.

2.2 Laws

In this section different laws are listed that are directly derived from the definitions stated above.

In the following x, y, z are boolean expressions.

2.2.1 Associativity

- $x\&(y\&z) = (x\&y)\&z$
- $x|(y|z) = (x|y)|z$

2.2.2 Commutativity

- $x \& y = y \& x$
- $x|y = y|x$

2.2.3 Distributivity

- $x \& (y|z) = x \& y|x \& z$
- $x|y \& z = (x|y) \& (x|z)$

2.2.4 Identity

- $x \& 1 = x$
- $x|0 = x$

2.2.5 Annihilator

- $x \& 0 = 0$
- $x|1 = 1$

2.2.6 Idempotence

- $x \& x = x$
- $x|x = x$

2.2.7 Absorption

- $x \& (x|y) = x$
- $x|(x \& y) = x$

2.2.8 Negative absorption

- $x \& ((\sim x)|y) = x \& y$
- $x|(\sim x) \& y = x|y$

2.2.9 Complementation

- $x \& (\sim x) = 0$
- $x|(\sim x) = 1$

2.2.10 Double negation

- $\sim (\sim x) = x$

2.2.11 De Morgan

- $\sim (x\&y) = (\sim x)|(\sim y)$
- $\sim (x|y) = (\sim x)\&(\sim y)$

2.2.12 Elimination

- $x\&y|x\&(\sim y) = x$
- $(x|y)\&(x|(\sim y)) = x$

CHAPTER 3

Development Guide

This document gives an overview of the code in *boolean.py*, explaining the layout and design decisions and some difficult algorithms. All used definitions and laws are stated in *Concepts and Definitions*.

Contents

- *Development Guide*
  - *Testing*
  - *Classes Hierarchy*
  - *Class creation*
  - *Class initialization*
  - *Ordering*
  - *Parsing*

3.1 Testing

Test *boolean.py* with your current Python environment:

```
python setup.py test
```

Test with all of the supported Python environments using *tox*:

```
pip install -r test-requirements.txt
tox
```

If *tox* throws *InterpreterNotFound*, limit it to python interpreters that are actually installed on your machine:

```
tox -e py27,py36
```

3.2 Classes Hierarchy

3.2.1 Expression

3.2.2 Symbol

3.2.3 Function

3.2.4 NOT

3.2.5 AND

3.2.6 OR

3.3 Class creation

Except for BooleanAlgebra and Symbol, no other classes are is designed to be instantiated directly. Instead you should create a BooleanAlgebra instance, then use BooleanAlgebra.symbol, BooleanAlgebra.NOT, BooleanAlgebra.AND, BooleanAlgebra.OR BooleanAlgebra.TRUE and BooleanAlgebra.FALSE to compose your expressions in the context of this algebra.

3.4 Class initialization

In this section for all classes is stated which arguments they will accept and how these arguments are processed before they are used.

3.4.1 Symbol

& `obj` (Named Symbol)

3.5 Ordering

As far as possible every expression should always be printed in exactly the same way. Therefore a strict ordering between different boolean classes and between instances of same classes is needed. This is defined primarily by the sort_order attribute.

3.5.1 Class ordering

`BaseElement` < `Symbol` < `AND` < `OR`

`NOT` is an exception in this scheme. It will be sorted based on the sort order of its argument.

Class ordering is implemented by an attribute `sort_order` in all relevant classes. It holds an integer that will be used for comparison if it is available in both compared objects. For Symbols, the attached *obj* object is used instead.

Class	sort_order
BaseElement	0
Symbol	5
AND	10
OR	25

3.5.2 Instance ordering

BaseElement `FALSE` < `TRUE`

`Symbol`

`Symbol.obj` o `Symbol.obj`

NOT if `NOT.args[0]` == `other` —> `other` < `NOT`

`NOT` o `other` —> `NOT.args[0]` o `other`

AND `AND` o `AND` —> `AND.args[0]` o `AND.args[0]`

if undecided: repeat for all args

if undecided: len(`AND.args`) o len(`AND.args`)

if undecided: return `AND` < `AND`

OR `OR` o `OR` —> `OR.args[0]` o `OR.args[0]`

if undecided: repeat for all args

if undecided: len(`OR.args`) o len(`OR.args`)

if undecided: return `OR` < `OR`

3.6 Parsing

Parsing is done in two steps: A tokenizer iterates over string characters assigning a TOKEN_TYPE to each token. The parser receives this stream of token types and strings and creates adequate boolean objects from a parse tree.

CHAPTER 4

Acknowledgments

1. Nicolaie Popescu-Bodorin: Review of "*Concepts and Definitions*"
2. Silviu Ionut Carp: Review of "*Concepts and Definitions*"